Safe reconfiguration of Coqcots and Pycots components

Jérémy Buisson, Fabien Dagnat, Elena Leroux, Sébastien Martinez

Safe reconfiguration of Coqcots and Pycots components

Jérémy Buisson[a,b,*], Fabien Dagnat[a,c], Elena Leroux[a,d], Sébastien Martinez[a,c]

[a] *IRISA, Brest – Rennes – Vannes, France*
[b] *St-Cyr Coëtquidan Schools, Guer, France*
[c] *Télécom Bretagne, Brest, France*
[d] *University of South Brittany, Vannes, France*

Abstract

Software systems have to face evolutions of their running context and users. Therefore, the so-called *dynamic reconfiguration* has been commonly adopted for modifying some components and/or the architecture at runtime. Traditional approaches typically stop the needed components, apply the changes, and restart the components. However, this scheme is not suitable for critical systems and degrades user experience. This paper proposes to switch from the stop/restart scheme to *dynamic software updating* (DSU) techniques. Instead of stopping a component, its implementation is replaced by another one specifically built to apply the modifications while maintaining the best quality of service possible. The major contributions of this work are: (i) the integration of DSU techniques in a component model; (ii) a reconfiguration development process including specification, proof of correctness using Coq, and; (iii) a systematic method to produce the executable script. In this perspective, the use of DSU techniques brings higher quality of service when reconfiguring component-based software. Moreover, the formalization allows ensuring the safety and consistency of the reconfiguration process.

Keywords: Dynamic reconfiguration, component model, dynamic software updating, DSU, Python, Coq, runtime evolution

1. Introduction

Software systems need to be highly available and should be built using secure, safe, performant, and robust components. These components must be regularly modified to fix vulnerabilities and bugs, to face new environments, and to offer new services. Enabling these evolutions, while maintaining a high level of availability, requires changing the architecture of such a system during

*Corresponding author
Email addresses: `jeremy.buisson@irisa.fr` (Jérémy Buisson), `fabien.dagnat@irisa.fr` (Fabien Dagnat), `elena.leroux@irisa.fr` (Elena Leroux), `sebatien.martinez@telecom-bretagne.eu` (Sébastien Martinez)

its execution. This capability is especially important for critical systems such as air-traffic control systems and networks, in which stopping systems is not an option due to financial or human costs. It also improves user experience as a user can continue to use a software system being updated without noticing the update, *i.e.*, the updating process is transparent to the users.

In software engineering, *dynamic reconfiguration* was introduced to build component-based software systems that can be modified during their execution, with minimal or no interruption. In this approach, components and connectors of a system can be inserted, removed or replaced at runtime, thus fostering the continuity of the provided services.

Since the proposition of the *quiescence* concept [1], reconfiguring a system typically requires the suspension of a set of components that will be affected by the reconfiguration. Maintaining the system in an operational status while stopping part of its components leads to a visible degradation of its quality of service [2, 3] due to component dependencies. Another essential issue to be considered is to preserve the consistency of the component assembly throughout the reconfiguration process. Some works in the literature have faced these challenges by minimizing the set of suspended components and/or decreasing the duration of their suspension [4, 5]. Focusing on consistency, Boyer et al. [6] propose a scheme in which an invariant requires to stop any component that depends on a stopped component. However, such dependencies often propagate up to the user frontend, and then the system may be almost entirely stopped.

To maintain service continuity, we must refrain from stopping components when reconfiguring a system. In this paper, we propose to use *dynamic software updating* (DSU) techniques [7, 8] instead of suspending components. The main idea is to mitigate the effect of any reconfiguration action by dynamically updating the implementation of directly and indirectly affected components. For example, if a component B used by a component A needs to be reconfigured, then the component A can be updated with a new implementation that no longer uses B before reconfiguring B. Once the reconfiguration involving B terminates, A may switch again its implementation either to fall back on the original behavior or to assume a new one better suited to the new configuration. Two important facts to notice are:

- The temporary update of A depends on the current state of execution and, therefore, is hard to foresee. Hence the *dynamic* update is needed.
- The better the temporary update hides the absence of B to the rest of the application, the more transparent the reconfiguration will be.

While changing the interconnection of the components seems easy and can possibly be generated by some algorithm, this small example highlights that dynamic reconfiguration incurs additional issues. The temporary behavior of A must be carefully designed such that application services are degraded as little as possible despite the absence of B. Whether this temporary behavior of A hides the absence of B or propagates part of the effects to the rest of the application greatly affects how the other components must be rearranged

too. Furthermore, this temporary behavior may need new components such as for example a component offering a service of B that A currently needs. By consequences, the temporary behavior of A can hardly be designed without the changes of architecture needed in mind, and conversely. In this context, producing a reconfiguration for an application consists in not only specifying the various components to create, the ones to modify and the ones to remove [6, 9, 10, 11] but also in specifying and implementing the various pieces of code required to maintain the activities of the application. When the architecture of the application becomes larger than a few components and connectors, the design and implementation of a reconfiguration become difficult. Using the example above, if the update of the component A is not finished when stopping B, the application may enter an inconsistent mode that may propagate to a user visible crash [12, 13]. Clearly, if a dynamic reconfiguration leads to a (visible) degradation of service, it becomes easier and equivalent to stop the application.

Our second idea is to guarantee the correct execution of a dynamic reconfiguration by proposing a complete reconfiguration design and implementation process that includes the use of a proof assistant to require a proof of the correctness of the reconfiguration. To reduce the cost of this development, this process is completely automated by tools ensuring the exchange of information between the running application, the proof assistant and the execution platform of the application. Using reflection, the current running architecture of the application is extracted and assertions describing this architecture are generated. Working in the proof assistant, the designer of the reconfiguration builds simultaneously the reconfiguration and its correctness proof. Once the designer is satisfied, the code of the reconfiguration is extracted from the proof and sent back to the platform executing the application. On receiving the reconfiguration script, the platform can apply it.

To validate practically this approach, we have designed a component model supporting DSU, *Pycots*. This component model is implemented in Python to reuse Pymoult[1] which is a DSU platform we have proposed in [14]. The component model includes a reflection level to extract the current running architecture and translate it into the chosen proof assistant, Coq[2]. This translation process uses an abstract version of the component model named *Coqcots* aimed at being the version the designer will use when building a reconfiguration.

The purpose of this paper is to report on this complete reconfiguration development process and describe precisely the various models and tools developed to support its automation. Therefore the contributions presented here will be:

1. A concrete component model *Pycots* to support execution and abstract component model *Coqcots* to support proving reconfiguration. A fully bidirectional translation is supported by a reflective feature of the concrete component model on one side and the extraction facilities of Coq for the other side.

[1] https://bitbucket.org/smartinezgd/pymoult
[2] http://coq.inria.fr/

2. An implementation of *Pycots* integrating DSU techniques.

3. A complete engineering process to help the design of a correct reconfiguration and its application to a running software. The enactment of this process is supported by tools automating the translation processes, Python code generation and the possibility to generate repetitive parts of the proof.

In this paper, Section 2 describes the domains involved in the presented work. Section 3 gives an overview of our proposal along with the supported reconfiguration scenarios. Section 4 describes our approach in details. Section 5 summarizes the main steps of the reconfiguration process. Section 6 compares our approach to related work. Section 7 concludes the paper with our main contributions and future directions.

All the material is available at `http://coqcots.gforge.inria.fr`.

2. Background

Before describing our contribution, we need to describe the three domains on which our proposal is built. First, our paper focuses on component-based applications and their reconfiguration. Then, we propose a new component model relying on mechanisms coming from the *Dynamic Software Update* community. At last, our methodology relies on the Coq proof assistant requiring the reader to understand the basics of Coq.

2.1. Component Architectures

Software components were proposed as a solution to increase productivity by promoting reuse to a large scale. Components foster reuse by enforcing a very low coupling between a component and its environment through the use of architecture as a key artifact during the development life cycle. In the early stage of the development, software architects use very abstract components without taking into account the physical underlying infrastructure. Later on, when the infrastructure is defined, these abstract concepts can be realised as actual software entities. This clean separation between functional architecture and code and the interaction with physical resources enables to build really reusable functional entities (both at abstract and code level).

The different concepts and their usage rules are defined through so-called *component models*. Many component models are currently available [15]. They can be either generic, such as CCM [16], SOFA [17] or domain specific such as PECOS [18]. They range from simple models relying on basic notion of component as UML2.0 [19] to very complex models atop of sophisticated infrastructures such as Fractal [20]. Component paradigm is built upon two main concepts. First, *components*, which encapsulate treatments and provide functionalities called *services* through access points named *ports*. Then, *architectures*, which describe an application as a set of components and their relations.

In this article, we use the component paradigm to be able to reconfigure a running application. Our focus is therefore on running component instances. Each link between components models a reference. This executable model, called Pycots, defines how our Python components are defined and executed. Reconfiguring an application in our approach consists in creating new component instances, removing existing instances and modifying the running code of existing components. The central concept to define such a representation is the architecture of the application with an abstract model describing the running instances and their dependencies. In our proposal, this abstract model is named Coqcots and is aimed at being manipulated with the proof assistant Coq. To depict such an abstract component architecture, we reuse the graphical representation of UML for depicting (Figure 2 on page 8 or 4 on page 10 for example). Following a common practice (see [20] for example), our executable model offers a reflection facility that supports the extraction of an abstract architecture representing a running application.

2.2. Dynamic Software Update

Updating applications is mandatory to apply bug fixes and vulnerability patches, and more generally to support software evolution. Generally, such updates require to stop the software, patch it and then restart it. This results in downtime and state / data loss, which are at best undesirable, sometimes costly, at worst unacceptable for critical systems. Following this observation, a large collection of mechanisms have been proposed to update software systems while they are running with no or little service interruption. To update dynamically an application, one has to handle different tasks such as transforming the data (*e.g.* adding or removing fields in objects, changing data representation) or rerouting the control flow (*e.g.* changing instructions in functions and methods). Many platforms addressing that issue can be found in the literature [7, 8] each of them using different techniques for applying dynamic updates. These techniques address the updates of elements of the program such as classes or functions, most of them do not aim to modify the architecture of the application.

This section presents the main tasks of a dynamic update and the most common techniques used for handling them. We also present the way Pymoult, the Python DSU platform used by Pycots, implements these techniques.

The first task is to detect the right moment for applying the update. The application has to be in a stable state where updating wouldn't make it enter an inconsistent state, *e.g.*, out of date function using updated data, because this may lead to a crash. The moment when the elements to be updated are quiescent is an example of right moment for applying an update. There are several ways to detect the right moment for an update. The detection can be: (1) static as in Kitsune [21] or Ginseng [22] where the application developer has to indicate a point in the program where the application is globally quiescent, *i.e.*, most of the elements of the application are quiescent, or (2) dynamic as in Hotswap [23] or ReCaml [24] where the virtual machine running the application can detect specific VM safe points that are proper for updating. The right

moment for updating can also depend on the update to apply. In Ksplice [25], a function can be updated only when it is not in the stack.

After the right moment for updating has been detected, we have to update the data and the control flow. Again, there are many ways to handle this two tasks. All the data can be transformed at once as in Kitsune or it can be transformed only when accessed, *e.g.*, a variable will be updated when a function reads from it, as in Ginseng. We call these methods eager and lazy data update. For updating the control flow, Hotswap and Ginseng redefine the functions and methods of the program, replacing the old functions by their new version. Ginseng introduces indirection using function pointers while Hotswap uses a specifically enhanced JVM for that purpose. ReCaml and Kitsune reboot the threads, making them run the new code while keeping the data. Both platforms allow the threads to start at a given point instead of rebooting from the beginning. ReCaml enables the stack to be reconstructed while Kitsune allows the execution of a rebooted thread to be guided for that purpose.

Pymoult provides these mechanisms among others and wraps them in a set of `Manager` classes. For example, an `EagerConversionManager` takes in charge the eager update of data. A manager instance is responsible of the choreography of a set of updates. A manager works in synergy with an `Update` class instance that supplies the information necessary for carrying out an update. For instance, suppose we want to update all the instances of a class `A` and an `EagerConversionManager` has been set up when developing the application. We can use an instance of `EagerConversionUpdate` to specify that we want to update the instances of the class `A` and describe the transformation that should be applied to them.

Using Pymoult, it is possible to develop custom `Manager` and `Update` classes by combining basic mechanisms provided by the platform through a low level API. For example, one can use the function `isFunctionInAnyStack` to check if a function can be updated safely as in Ksplice.

2.3. Proof Assistant

Based on formal calculi that formalize the notions of proposition and proof, proof assistants are software that help proving theorems. They mechanically verify proofs according to the rules of an underlying formal calculus. The proof assistants usually provide environments for interactive proof development that embed decision procedures to generate automatically some parts of the proofs.

A typical approach to proof assistants consists in relying on the *Curry-Howard isomorphism*, which states that the relation between a proof of a proposition and this proposition is the same as the relation between a λ-term and its type. For instance, the *modus ponens* deduction rule corresponds to the typing rule for function application; the rule for implication introduction corresponds to the typing rule for λ-abstraction. Following this approach, verifying a proof is the same task as type-checking a λ-term.

Coq is such a proof assistant based on the *calculus of inductive constructions*, a typed λ-calculus with dependent types. Furthermore, type expressions are

```
Definition x: (nat → bool) →
nat → bool := fun a b ⇒ a b.
```

```
Definition x: (nat → bool) →
nat → bool.
  intros a b.
  apply a.
  apply b.
Defined.
```

Figure 1: A Coq definition with the expression language (left) and proof-mode (right).

first-class expressions, hence types can be used like any other value. Intuitively, it allows the type of the return value of a function to depend on the effective value of the parameters. A typical example for dependent types is the type of the C-like `printf` function: the types of the to-be-formatted parameters depend on the content of the formatting string. Dependent types can also convey correctness properties, which we use in this paper to express preconditions and postconditions in the abstract component model Coqcots.

Since Coq proofs are λ-terms, they are executable and conversely programs contain proofs. Coq defines two types of types to make an explicit distinction between: `Prop` for logical types, i.e., propositions; and `Set` for computational types. Based on this distinction, the *extraction* mechanism automatically translates the computational `Set` parts of a program / proof to a standard programming language, while it leaves the logical `Prop` parts out. This approach provides an engineering process to develop formally verified programs, like illustrated by the CompCert project [26].

To define proofs and programs, Coq comes with two languages. Functional expressions follow a syntax close to usual functional programming languages; while the proof mode lets build an expression by the means of sequentially applying the rules of the calculus of inductive constructions. Figure 1 shows side-by-side the two languages for the same example, where `intros` is implication introduction and `apply` is the modus ponens rule.

To automate (parts of) the proofs, Coq introduces *tactics*, which can be considered as macros. It is therefore possible to program procedures that automatically apply rules and other tactics during a proof step.

3. An overview of our proposal

The objective of this section is to give an overview of our process to build correct reconfiguration. This process is illustrated by the example shown on Figure 2. The upper left part of this figure presents a part of an architecture. This part is composed of six components A, B and C_1 to C_4. Each component C_i (where $i = [1..4]$) uses a service of the component A. These four dependencies will be used to illustrate some of the possible situations encountered when reconfiguring a real application. Notice that all these components may also be used by or use other components not represented here. The only constraint is that no other component uses a service offered by A.

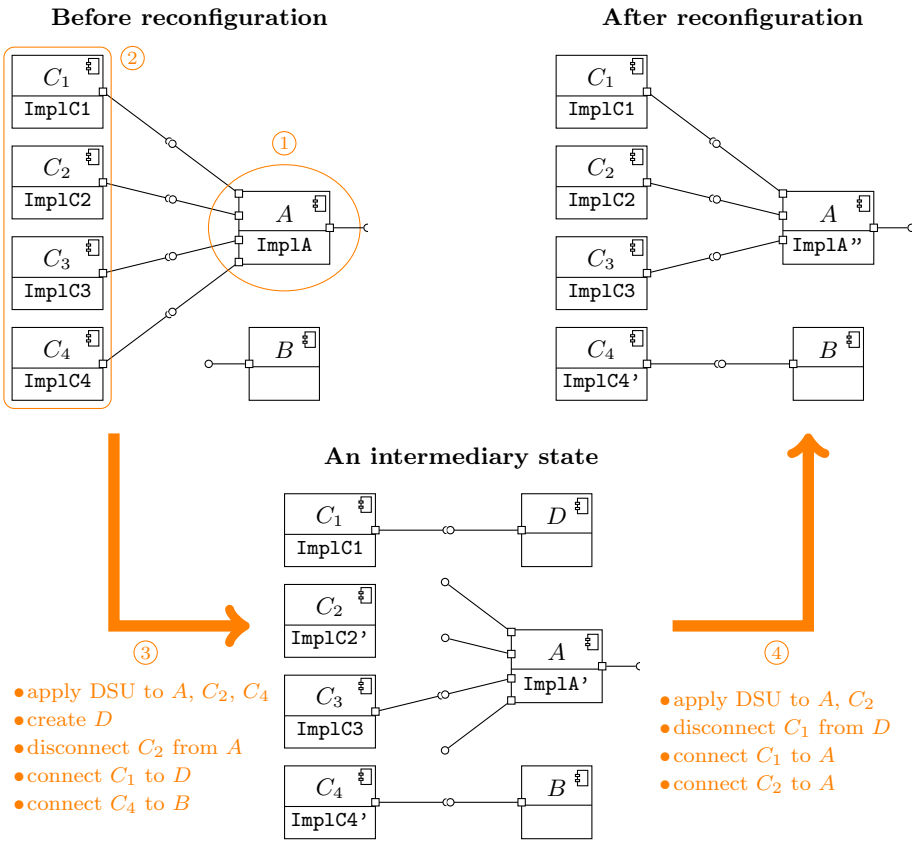

Figure 2: Architecture of a part of an application before, during and after a reconfiguration.

Suppose that we need to reconfigure the component A in order to remove one of the services it provides. For example, the fourth port used by C_4 will be removed. The first step is to detect the set of components to be reconfigured. In our example this set will contain the component A (see ① of Figure 2). [3] Once A is found, the designer of the reconfiguration must identify all components using its services. For our example we obtain the set containing the components from C_1 to C_4 (see ② of Figure 2). Then, we must decide how we are going to hide to the rest of the application the fact that A is going to be only partly available. For our example, we have decided to:

1. Use a transient stub: keep C_1 unchanged, create a new transient component D, and connect C_1 to this new component D instead of using A.

[3] For real reconfiguration finding the set of components to modify may be a little harder.

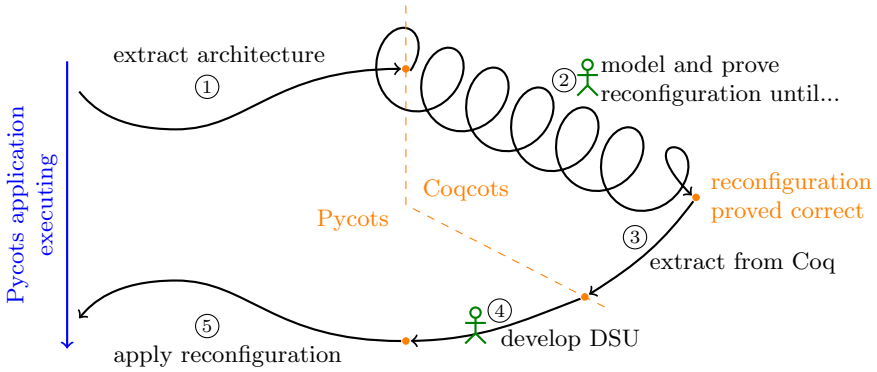

Figure 3: Overview of the proposed approach.

2. Propagate service degradation to the client: modify the implementation of C_2 to make it independent of A permitting to disconnect C_2 from A.

3. No impact on the client: keep C_3 exactly the same, it still uses A.

4. Switch to a new provider: update the implementation of C_4 to make it use the existing component B instead of A.

Making such design decisions usually requires human intervention. In step ③, we have to realize all the operations needed to support the previous choices. Each modification of the implementation is done using DSU mechanisms. We also have to update the implementation of A to prepare it for the intended modification, *e.g.*, stopping some threads. Notice that A cannot be completly stopped here as C_3 is still using it (`ImplA'` has to take it into account).

Finally, A can be updated as intended and the various temporary operations may be reverted (see ④ of Figure 2). Here for example, we have chosen that: (i) C_1 is reverted to using A; (ii) C_2 is reverted to its previous behavior using A; (iii) C_3 is still unchanged; and (iv) C_4 will continue to use the component B.

The example concretizes the process of reconfiguring an application and illustrates the fact that a reconfiguration script is a complex piece of program. To help the designer of such a reconfiguration, we propose a process including the proof assistant Coq to foster the construction of correct reconfigurations. This process is depicted in Figure 3. The left arrow represents the execution flow of the application we need to reconfigure. When a reconfiguration is needed, its design and execution follow the five steps on the right of the figure. Steps ② and ③ of this process are performed in Coq using our abstract component model, Coqots. The other three steps use our concrete component model Pycots.

① The current architecture of the target software system is extracted using the reflexive feature of the Pycots execution platform. The result of this operation is a Coq module containing a Coqcots architecture. This part of our process is described in details in subsection 4.1 on page 11.

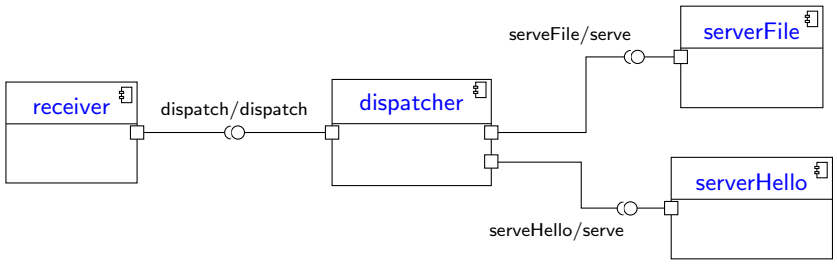

Figure 4: The initial software architecture of the web server.

② The designer works to define the reconfiguration and build the proof of its correctness within Coq as described in subsection 4.2 on page 15. Like previously illustrated by the example of Figure 2, this step can involve the design and creation of new components and new implementations. Human intervention is mandatory to carry out this step.

③ Once the reconfiguration has been proved, the reconfiguration script is extracted from this proof using Coq facilities (see subsection 4.3 on page 21).

④ All the glue code need to be developed in Python. The various DSU pieces of code must be implemented as described in subsection 4.4 on page 22.

⑤ Finally, the Python (reconfiguration) script is submitted to the Pycots manager, which is a platform component that receives and applies reconfiguration scripts to the target software system. This last step is detailled in subsection 4.5 on page 24.

4. The complete process in details

In Section 3 we informally described the main steps of our reconfiguration approach shown on Figure 3. The purpose of this section is to provide the details of this reconfiguration development process. To illustrate each of its steps, we use a simple web server case study. Figure 4 contains the initial architecture of our web server which contains four components: (1) the receiver wraps an instance of `BaseHTTPServer`, which receives and decodes HTTP requests; (2) the dispatcher dispatches requests to handlers according to the requested URL; (3) serverHello generates a dynamic web page with a "Hello, world" greeting, and; (4) serverFile detects that the given URL is a file name whose content is sent as a response. The purpose of this case study is twofold. First, it aims at explaining concretely our process. And second, it demonstrates that our approach enables the continuity of service of the web server without stopping it during the reconfiguration process.

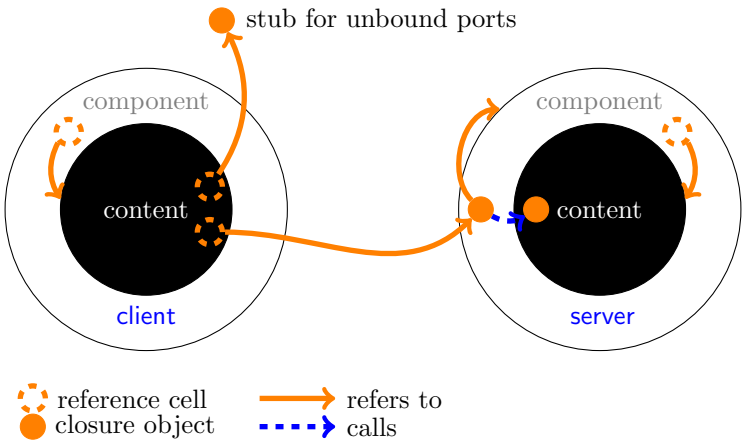

Figure 5: Anatomy of a simple client server Pycots architecture.

4.1. Generating Coqcots architecture from a Pycots execution state

The first step of the process, which is shown as ① in Figure 3 on page 9 consists in introspecting the execution state according to the Pycots framework in order to generate a reified architecture in Coqcots. For the sake of simplicity, in this subsection we use a client-server example instead of the web server whose Pycots and Coqcots architectures are more complex and, therefore, more difficult to explain. In the first two parts of this subsection we describe the client-server architecture using respectively the Pycots and the Coqcots formalisms. Then, we explain how to realize the introspection of the Pycots architecture. To close this subsection, we briefly present an extract of the Coqcots architecture for our web server case study.

Overview of Pycots. In Pycots, each component is a black-box content object wrapped in a component object, as depicted in Figure 5. Each wrapper component object has a reference to its wrapped content object. The granularity level for dependencies is the method level. Provided methods are methods implemented by the content object for which a proxy (implemented by closure) is provided by the component object. Dependencies are reference cells injected by the framework in the content object: when bound, the reference cell refers to a proxy of another component; when unbound, the reference cell refers to a stub provided by the framework.

In this context, programming a component amounts to providing the class of its content object (`Client` or `Server` in the code below). This class must include the code of the provided ports (`Server` defines the method `echo_port`). Finally, the developer must use functions provided by the framework to create the component (`pycots.create`) using its implementation class, the list of its provided ports and the list of its required ports. Once at least two components are created, it is possible to bind them using `pycots.bind`.

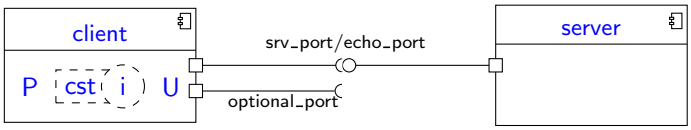

Figure 6: Anatomy of a simple client server Coqcots architecture.

```
import pycots
class Client(object):
    def __init__(self): super(object, self).__init__()
    def run(self): self.srv_port()
class Server(object):
    def __init__(self): super(object, self).__init__()
    def echo_port(self): print "hello"
c = pycots.create(Client, [], ["srv_port","optional_port"])
s = pycots.create(Server, ["echo_port"], [])
pycots.bind(c, "srv_port", s, "echo_port")
pycots.call(c, "run")
```

Coqcots architectures. Figure 6 depicts the same architecture as Figure 5 with the Coqcots point of view. As illustrated, a component (client) has an associated implementation (i) that uses a set of *used services* (U, which contains only the srv_port and optional_port services in the case of the client component) to provide a set of *provided services* (P, which is empty for the client component). The architecture instance contains the client and server components and defines the binding between the used service of the client and the provided service of the server (srv_port/echo_port). When a used service of a component is not bound, the implementation of that component cannot use this unbound service.

In our approach, the implementation of a component may come with a precondition that specifies its assumption on the architecture in which the component is instantiated. This *architectural constraint* (cst for the client component) is an invariant that ensures that the implementation will only run in the expected situation. For example, a given component may require that some of its used services must be bound (the *mandatory services*), while the other services are allowed to become unbound (the *optional services*). The component implementation can then assume that only optional services may be unbound. Using architectural constraints enables the designer to assume the mandatory semantics discussed by Boyer et al. [6] and Bruneton et al. [27].

Formally, Coqcots comes with two predicates, *contains* and *binds*, which respectively state that an architecture contains a given component and a given binding. The *contains* predicate takes as first argument an architecture a, followed by the five elements of a component: (1) the name of the component c,

(2) the set[4] of its used services U, (3) the set of its provided services P, (4) its architectural constraint cst, and (5) its implementation i. Its definition is:

```
Parameter contains:
    ∀ (a: arch) (c: comp) (U: facet) (P: facet) (cst: arch → comp → Prop)
      (i: ∀ (u: facet_record U), cst a c → no_exc_if_bound a c u → facet_record P),
      Prop.
```

The architectural constraint is a function that maps an architecture a and a component c to a proposition. It is used by the implementation as a precondition to the component's services (term cst a c). The other precondition no_exc_if_bound relieves the implementation from defensively checking for availability when a used service is guaranteed to be bound according to the constraint cst.

The *binds* predicate formally states that a binding exists in an architecture a. The binding is represented by six values: three for the client component (the user of the binding) and three for the server component (the provider of the binding). Both components are represented by (1) their identity respectively clt and srv, (2) the set of their involved services clt_U and srv_P, and (3) the ports clt_port and srv_port bound by the predicate. The definition of *binds* is:

```
Parameter binds:
    ∀ (a: arch) (clt: comp) (clt_U: Type) (clt_port: namedport clt_U)
      (srv: comp) (srv_P: Type) (srv_port: namedport srv_P),
      Prop.
```

Using these predicates, the designer can define an architecture. For example, the architecture presented in Figure 6 is an element of the following type:

```
Definition client_server :=
  { a | ∃ client server,
      contains a client client_use_facet client_provide_facet
          client_constraint (client_implementation a client)
      ∧ contains a server server_use_facet server_provide_facet
          server_constraint (server_implementation a server)
      ∧ binds a client _ srv_port server _ echo_port }.
Definition client_constraint (self_arch: arch) (self: comp) :=
      ∃ s provs port, binds self_arch self _ srv_port s provs port.
Definition server_constraint (self_arch: arch) (self: comp) :=True.
```

In this example, the client srv_port must be bound as it is a mandatory dependency, and the server has no constraint. For the binding, the set of used and provided ports (third and sixth arguments elided as _) are inferred by Coq.

Coqcots invariants. Coqcots is equipped with a set of invariants to ensure the soundness of the architecture definition:

- Correct typing of bindings. This invariant excludes all architectures containing at least two ports bound together having incompatible types.
- Existence of bound components. This invariant checks that bound components belong to the architecture.

[4] In our model, we define a set of ports as a **facet**.

- Unicity of used service bindings. This invariant checks that the architecture does not contain a used port bound to two different provided ports.[5]
- Unicity of components characteristics. This invariant ensures that, for each component, the set of ports, constraints, and implementation are defined only once.[6]
- Satisfaction of component constraints. This invariant ensures that for each component, the architectural constraint given in its definition holds.

An architecture is *consistent* if the five previously defined invariants hold. We have proved that any architecture, obtained by applying Coqcots reconfiguration operations starting from the empty architecture, is consistent. It relies on two sub-proofs: (1) the empty architecture is consistent and (2) any of the five proposed reconfiguration operations (detailed in Section 4.2) preserves the invariants and then consistency. The length of the proofs is about 2500 lines[7].

Introspection in Pycots. The introspection of the architecture of a running Pycots application relies on the native facilities of Python. Namely, for each component, the methods and fields of the wrapper and content objects are scanned. Regarding the wrapper, the code object[8] of each method is compared to the one of the proxy to detect the methods that are provided by the component. Regarding the content, the code object of each method is compared to the one of the proxy as well as the one of the stub. In the former case, a bound dependency is detected, and the binding can be introspected in the values captured by the closure object of the proxy. In the latter case, an unbound dependency is detected. The whole architecture is traversed following the bindings from an initial set of root components.

The extracted architecture of the web server. Finally, we automatically apply the introspection method described above to the web server Pycots architecture in order to obtain its reified Coqcots architecture shown on Figure 4. An extract of this architecture expressed as a Coq definition is given below. For the sake of simplicity, only an extract is presented here.

[5] This invariant restricts the component model to $1-1$ bindings. To enable other cardinalities, one can either remove this invariant, use reconfiguration to create or remove ports on need like, e.g., Fractal's collection interfaces, or introduce communicating elements, e.g., connectors responsible to balance, broadcast, gather or scatter communications.

[6] Despite this invariant seems to state the obvious, without it, Coq would not prevent from writing a proposition stating that a component has two different definitions. This invariant is also useful in proofs as it allows to deduce the equality of two component definitions: when the differences are manifest the invariant can be used to complete the proof by contradiction; otherwise, one definition can be replaced with the other one in other hypotheses and/or proof goals in order to make the proof progress.

[7] The proof script is in the `coqcots/Consistency.v` file of the source code.

[8] In Python, the instructions of a function, closure or method are wrapped in a code object.

```
Definition architecture := { a & { dispatcher & { receiver & { serverFile & { serverHello |
(* list of all the components *)
(∀ c U P (C: arch → comp → Prop) (i: ∀ a c u, C a c → no_exc_if_bound a c u → _),
   contains a c U P C (i a c) → c=dispatcher ∨ c=receiver ∨ c=serverFile ∨ c=serverHello)
(* all the components are distinct *)
∧ dispatcher ≠ receiver
(* existence component dispatcher *)
∧ contains a dispatcher dispatcher__usefacet dispatcher__provfacet
            dispatcher__constraint (dispatcher__impl a dispatcher)
(* characteristics of component dispatcher *)
∧ (∀ U P (C: arch → comp → Prop) (i: ∀ a c u, C a c → no_exc_if_bound a c u → _),
     contains a dispatcher U P C ( i a dispatcher) → U = dispatcher__usefacet )
∧ ( (* ... *) P = dispatcher__provfacet ) ∧ ( (* ... *) C = dispatcher__constraint )
∧ (∀ i, contains a dispatcher dispatcher__usefacet dispatcher__provfacet
                  dispatcher__constraint (i a dispatcher) → i = dispatcher__impl)
(* existence of the binding from receiver to dispatcher *)
∧ binds a receiver dispatch0 dispatcher dispatch
(* list of all the bindings (excerpt) *)
∧ (∀ c ct cp s st sp, binds a c ct cp s st sp → c=dispatcher ∨ c=receiver)
∧ (* ... *)
```

The complete description for the architecture of Figure 4 is the conjunction of 44 such facts. It can be found at `http://coqcots.gforge.inria.fr/demojss/coqcots-coqdoc/Webserver.html`.

4.2. Developing a proved reconfiguration

In the previous subsection we explained how to introspect the execution state of a system using Pycots and how to obtain a reified Coqcots architecture. The purpose of this subsection is to explain how to realize the reconfiguration of this architecture (the step ② in Figure 3 on page 9). For this purpose we need to introduce several primitive reconfiguration operations as well as some proof patterns in order to build and to validate the reconfiguration. Developed reconfigurations should be correct with respect to the preconditions expected by reconfiguration operations, as well as to the constraints required by the components. To illustrate the reconfiguration operations, we describe how the reconfiguration leading to Figure 7 on the next page involves the operations we define.

Reconfiguration operations. The five primitive reconfiguration operations, which we formally define in this section, are:

1. *create* adds a new component to the current architecture by taking its used and provided ports, constraint and implementation.
2. *destroy* removes an existing component from the current architecture by taking the name of the component.
3. *link* creates a binding from a used port of a component to a provided port of another component by taking the requiring component and its used port and the providing component and its provided port.

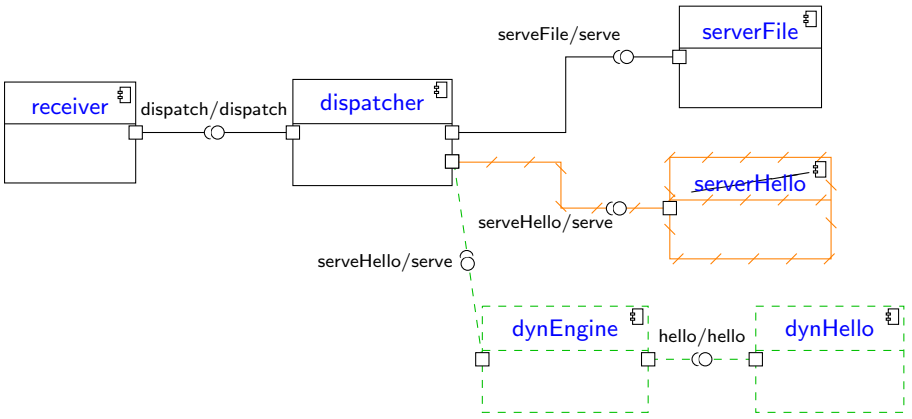

Figure 7: The software architecture of the web server after dynamic reconfiguration.

4. *unlink* destroys a binding from the current architecture by using the same parameters as *link*.

5. *hotswap* changes the behavior of an existing component by taking the component's name, the four new elements of the component, and two functions mapping respectively the used ports and the provided ports of previous version to the ones of the new version.

In following, we first explain in details the *create* operation, whose Coq code is quite compact and easy to understand. Then we give a bief description of the *hotswap* operation and we underline the importance of this operation in our reconfiguration process. We omit the descriptions of all the other operations as they can be presented in the similar way as the *create* operation.

The *create* operation is a function that returns a pair r composed of the new architecture new_a and the newly created component new_c. They satisfy the create_post postcondition described later in this section. The *create* function takes seven parameters: (1) the current architecture a, (2) the set of used services of the new component U, (3) a proof U_all_opt that the used services are all of the type **optional**[9], (4) the set of provided services P, (5) the constraint cst, (6) the implementation i, and (7) a proof cst_all_hold that in the resulting architecture the architectural constraints of all the components (including the created one) are satisfied. The *create* operation is defined by:

```
Parameter create:
  ∀ (a: arch) (U: facet)
    (U_all_opt: List.Forall (fun p ⇒ ∃ t, p = optional t) (ports_of _ (facet_spec U)))
    (P: facet) (cst: arch → comp → Prop)
    (i: ∀ self_arch self u, cst self_arch self
```

[9]The **optional** type models that used services can be unbound, thus having no value.

```
      → no_exc_if_bound self_arch self u → facet_record P)
  (cst_all_hold: ∀ new_a new_c, create_post a U U_all_opt P cst i new_a new_c
      → ∀ c' U' P' (cst': arch → comp → Prop)
          (i': ∀ a'' c'' u, cst' a'' c'' → no_exc_if_bound a'' c'' u → _),
      contains new_a c' U' P' cst' (i' new_a c') → cst' new_a c'),
  { r: arch × comp | let (new_a, new_c) := r in
    create_post a U U_all_opt P cst i new_a new_c }.
```

The postcondition of the *create* operation is the conjunction of six parts: (1) The previous architecture a does not contain the newly created component new_c. (2) The new component new_c exists in the new architecture new_a with the given elements (used U and provided P services, constraint cst and implementation i). (3) The new architecture new_a contains all the components of the previous architecture a. (4) The new architecture new_a contains only the components contained by the previous architecture a and the new component new_c. (5-8) In the new architecture new_a, the new component new_c is well-defined, *i.e.* it has unique elements (U, P, cst and i). (9-10) The previous a and new new_a architectures contain exactly the same bindings.

```
Definition create_post
  (* same parameters as create *) (new_a: arch) (new_c: comp) :=
1  (∀ U' P' cst' i', ¬ contains a new_c U' P' cst' i')
2  ∧ (contains new_a new_c U P cst (i new_a new_c))
3  ∧ (∀ c' U' P' (cst': arch → comp → Prop)
       (i': ∀ a'' c'' u, cst' a'' c'' → no_exc_if_bound a'' c'' u → facet_record P'),
       contains a c' U' P' cst' (i' a c') → contains new_a c' U' P' cst' (i' new_a c'))
4  ∧ (∀ c' U' P' (cst': arch → comp → Prop)
       (i': ∀ a'' c'' u, cst' a'' c'' → no_exc_if_bound a'' c'' u → facet_record P'),
       contains new_a c' U' P' cst' (i' new_a c')
       → c' = new_c ∨ contains a c' U' P' cst' (i' a c'))
5  ∧ (∀ U' P' cst' i', contains new_a new_c U' P' cst' i' → U' = U)
6  ∧ (∀ U' P' cst' i', contains new_a new_c U' P' cst' i' → P' = P)
7  ∧ (∀ U' P' cst' i', contains new_a new_c U' P' cst' i' → cst' = cst)
8  ∧ (∀ (i': ∀ a'' c'' u, cst a'' c'' → no_exc_if_bound a'' c'' u → facet_record P),
       contains new_a new_c U P cst (i' new_a new_c) → i' = i)
9  ∧ (∀ clt clt_U clt_port srv srv_P srv_port,
       binds a clt clt_U clt_port srv srv_P srv_port
       → binds new_a clt clt_U clt_port srv srv_P srv_port)
10 ∧ (∀ clt clt_U clt_port srv srv_P srv_port,
       binds new_a clt clt_U clt_port srv srv_P srv_port
       → binds a clt clt_U clt_port srv srv_P srv_port).
```

It is important to notice that a reconfiguration operation should preserve constraints and bindings of unaffected components. We address this point with the so-called *frame axiom* approach [28] in postcondition of reconfiguration operations. For example, regarding the *create* operation, postconditions 3, 4, 9 and 10 above are the frame axioms.

The *hotswap* operation is the key point of Coqcots. The goal of this operation is to replace ports, constraints and implementation of a given component. It is important to notice that all of these replacements must be done at once. This constraint of our model comes from the fact that the type of an implemen-

tation depends on ports and constraints. For example, changing a port while keeping the implementation unchanged, is not well-typed. For the same reason, the bindings must also be adjusted. To do so, the *hotswap* operation takes two additional parameters map_U and map_P, which map bound ports of the old set of ports to the new set of ports. As these mappings are restricted to bound ports (hypothesis p_bound in the definitions of portmap_u and portmap_p), it is possible to remove ports as long as they are not bound in the architecture. Our definition of the *hotswap* operation is therefore consistent with *contextual substitutability* as defined by Brada [29]. Port mappings are required to map old ports to distinct new ports (map_U_injective and map_P_injective); the mapping must be independent of its p_bound parameter, *i.e.*, the mapping must not depend on the structure of the proof that the port denoted by parameter p is bound[10] (map_U_proof_irrel and map_P_proof_irrel); the returned port must be in the new facet (map_U_valid and map_P_valid); and the types of the mapped ports must remain unchanged accross the mapping (map_U_preserve and map_P_preserve).

Note that, for the sake of space, in the two calls of hotswap_post, we omit the 21 parameters of *hotswap* which are passed as is.

```
Definition portmap_u (a: arch) (c: comp) (new: Type) (old: Type) :=
  ∀ (p: namedport old) (p_bound: ∃ srv st stp, binds a c old p srv st stp),
    namedport new.

Definition portmap_p (a: arch) (c: comp) (new: Type) (old: Type) :=
  ∀ (p: namedport old) (p_bound: ∃ clt ct ctp, binds a clt ct ctp c old p),
    namedport new.
Parameter hotswap:
  ∀ (a: arch) (c: comp) (prev_U: facet) (prev_P: facet)
    (existed: ∃ C i, contains a c prev_U prev_P C i)
    (new_U: facet)
    (new_U_all_opt: List.Forall (fun p ⇒ ∃ t, p = optional t)
                                (ports_of (facet_spec new_U)))
    (map_U: portmap_u a c (facet_record new_U) (facet_record prev_U))
    (map_U_injective: ∀ p1 p2 pb1 pb2, map_U p1 pb1 = map_U p2 pb2 → p1 = p2)
    (map_U_proof_irrel: ∀ p pb1 pb2, map_U p pb1 = map_U p pb2)
    (map_U_valid: ∀ p pb, List.In (map_U p pb) (facet_spec new_U))
    (map_U_preserve: ∀ p pb, np_type (map_U p pb) = np_type p)
    (new_P: facet)
    (map_P: portmap_p a c (facet_record new_P) (facet_record prev_P))
    (map_P_injective: (* ... *)) (map_P_proof_irrel: (* ... *))
    (map_P_valid: (* ... *)) (map_P_preserve: (* ... *))
```

[10] Since preconditions are given as parameters, Coq considers that the return of a function may depend on the structure of the proof of its preconditions. In the case of map_U and map_P, this behavior prevents from proving the Coqcots invariants (page 13) as well as the coalesced hypotheses (page 19). The Coq standard library comes with a general axiom of proof irrelevance inherited from the classical logic. Here we prefer to require its specific cases (map_U_proof_irrel and map_P_proof_irrel, and therefore it's up to the implementers of map_U and map_P decide whether they can prove the property in that specific case, or rely on the general proof irrelevance axiom.

```
(new_cst: arch → comp → Prop)
(new_i: ∀ self_a self_c (u: facet_record new_U), new_cst self_a self_c
        → no_exc_if_bound self_a self_c u → facet_record new_P)
(new_cst_all_hold: ∀ new_a, hotswap_post (* the parameters of hotswap *) new_a
        → ∀ c' U' P' (cst': arch → comp → Prop)
            (i': ∀ a'' c'' u, cst a'' c'' → no_exc_if_bound a'' c'' u → _),
        contains new_a c' U' P' cst' (i' new_a c') → cst' new_a c'),
{ new_a : arch | hotswap_post (* the parameters of hotswap *) new_a }.
```

The *hotswap* operation replaces the classical start/stop operations. Using this operation, the developer is able to offer a better continuity of service during a reconfiguration. Indeed, the developer can define a new behavior for the component providing partial services or all of its services by using other providers for its used services. Notice that this is possible even if the component initial design has not anticipated the situation. Last but not least, behavioral changes must consistently reflect in the type of the component, so that any service degradation is explicit in the component type. Hence, it is possible to check whether the other components still meet their quality of service requirements. This makes service degradation controllable.

Proof patterns. Using the frame axiom approach in the postconditions of the operations (as illustrated in create_post) makes their use inconvenient. Indeed, the proof environment accumulates the successive states of the architecture, with hypotheses linking each state to its predecessor state. Therefore any proof on the last state of architecture involves all the preceding states: the length of the proofs increases with the number of steps in the reconfiguration. To avoid this issue, we propose to wrap the reconfiguration operations in tactics, which automatically coalesce the proof environment such that it no longer refers to previous states of the architecture. The proof environment thus looks like the initial architecture given in Section 4.1 on page 15.

To illustrate this, consider the *create* operation. After the operation has been applied, the proof environment contains the two following hypotheses:

- From the previous architecture, prev_H states that (see ① of Figure 8 on the next page) if a component c' is contained in the previous architecture prev_a, it is one of the components previously created (here they are denoted c_1 to c_n).
- From the frame axiom, frame states that (see ② of Figure 8), in the new architecture new_a, any component is either the newly created component new_c or it was already contained in the previous architecture prev_a.

The tactic coalesces these two hypotheses into the equivalent single one presented at the bottom of the Figure 8 on the following page. Notice that ③ is now the combination of ① and ② and no longer refers to the previous state prev_a of the architecture. This coalesced hypothesis can be proved by the following proof script:

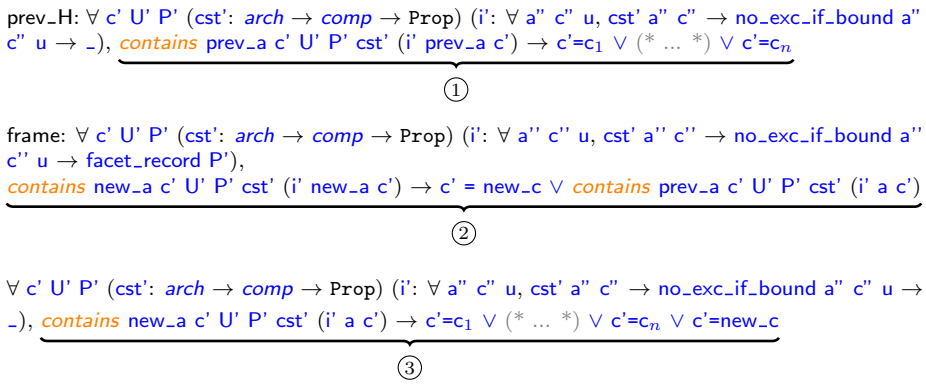

Figure 8: Two hypotheses and their coalesced form.

```
intros c' U' P' cst' i' new_a_contains_c'.
destruct (frame _ _ _ _ _ new_a_contains_c') as [ H | H ].
- auto.
- destruct (prev_H _ _ _ _ _ H); auto.
```

For each operation, we define the coalesced hypotheses such that the previous state of the architecture is no longer referred to. As all of these coalesced hypotheses can be systematically proved, we developed a tool to generate automatically the tactics that apply the operations then coalesce the proof environment.

Reconfiguration of the web server. The reconfiguration depicted in Figure 7 on page 16 splits the serverHello component into two components: dynEngine, a generic engine that generates dynamic pages, and dynHello, the greetings handler. Since the serverFile component is not affected by this reconfiguration, it will continue to handle requests during the reconfiguration. The main idea is to temporarily hotswap the implementation of the dispatcher component such that it continues to serve the requests targeting the serverFile component while it enqueues those for serverHello. Since this step is manual in the process (see Figure 3 on page 9), the reconfiguration developer can decide to design this new implementation. But if she/he attempts to use *hotswap* on dispatcher without any preparation, she/he would be unable to prove that the binding between receiver and dispatcher remains well typed, as requested by Coq as part of the proof obligations. Indeed this binding would not be well typed anymore. This error highlights that either receiver must be changed to accomodate the service provided by the new implementation of dispatcher, or the change of dispatcher must be reconsidered such that the type of its dispatch port is not modified. In our example, we choose the former fix: we modify receiver. The main steps of this reconfiguration are:

1. The implementation of the receiver component is modified using *hotswap* such that it spawns a new thread for each request. Therefore, it will be

possible to suspend a thread to delay a request while the other requests are served with no delay.

2. Using *hotswap*, the implementation of the dispatcher component is replaced by the following one: (1) it suspends the current thread (the request-handler thread) by using a global event object, if it receives a request for the serverHello component, and (2) it works as before, if a request for the serverFile component is received.

3. Once the two previous steps are completed, the serveHello port of dispatcher is no longer used: the web server is ready for the architectural changes. The binding between the dispatcher and serverHello components is removed (*unlink*) then the component serverHello itself is removed (*destroy*). The dynEngine and dynHello components are instantiated (*create*) and bound (*link*), and then dynEngine is (*hotswap*) to add its provided port. Last dispatcher is bound (*link*) to dynEngine.

4. Lastly, dispatcher is (*hotswap*) back to its initial implementation and suspended threads are resumed.

The definition of this reconfiguration contains about 200 lines. It proves that the reconfiguration is correct and that requests targeting the serverFile component are handled immediately, even during reconfiguration. The complete reconfiguration script is available online at: `http://coqcots.gforge.inria.fr/demojss/coqcots-coqdoc/Reconfiguration.html`.

4.3. Extracting the reconfiguration script

Once we have obtained a valid Coqcots reconfiguration, we need to transforme it back to an executable Pycots script (step ③ of Figure 3 on page 9).

The Coq `extraction` plugin translates the computational parts of Coq definitions while leaving logical parts out. This plugin proceeds first to an intermediate language MiniML, which is a variant of the λ-calculus with a fixed-point combinator, inductive and coinductive types, pattern matching, and a module and functor system. Then several backends generate concrete code for OCaml, Haskell and Scheme.

In this section, we present a new backend we have developed to target the Python language. The main problems to overcome in this backend concern (co)inductive types and pattern matching, which are missing features in the Python language.

Inductive and coinductive types. The approach we follow is based on the Scott encoding of data types into the λ-calculus [30]. With this approach, each inductive type defined by n constructors denoted $\{C_i\}_{i=1}^{n}$, where each constructor C_i has arity a_i, is translated to the functions:
$\{\lambda x_1 \ldots \lambda x_{a_i} \lambda c_1 \ldots \lambda c_n \cdot c_i x_1 \ldots x_{a_i}\}_{i=1}^{n}$.
For instance, the **list** type defined by:

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A → list A → list A.
```
is translated to the following Python functions:

```
nil = lambda if_nil, if_cons: if_nil()
cons = lambda head, tail, if_nil, if_cons: if_cons(head, tail)
```

Each function has the parameters of the corresponding constructor (none for `nil` and `head` and `tail` in the case of `cons`) plus one functional parameter for each constructor of the type (`if_nil` and `if_cons` in this case). As result, the function calls the functional parameter corresponding to the constructor it encodes (`nil` calls `if_nil`; `cons` calls `if_cons`) with the parameters of the constructor. With this encoding, a value is a closure resulting from the partial application of the function, where the `if_xxx` parameters are not supplied. The closure stores the effective parameters of the constructor.

Pattern matching. Following Scott encoding of inductive types, pattern matching is encoded as a function call to the matched object (which is encoded as a closure). All the cases are rearranged in a decision tree, such that each test is performed at most once.

For instance, the following definition:

```
Fixpoint count (A: Type) (l: list A) :=
  match l with
  | cons _ t ⇒ 1 + count _ t
  | nil ⇒ 0
  end.
```
is translated to the following Python function:

```
count=lambda l: l(lambda: 0, lambda _,t: 1+count(t))
```

The matching on list `l` is implemented by a call to the closure encoding this list. Each case is provided as a function whose parameters are the parameters of the constructor, *i.e.*, no parameter in the `nil` case and the head and tail of the list in the `cons` case.

4.4. Develop DSU and glue the reconfiguration script to the application

At step ④ of Figure 3 on page 9, the Python script extracted from the Coq code needs to be glued with the Pycots framework, as well as concrete Python objects (components, implementation objects and facets). The approach we follow is to wrap the script in a Coq functor, such that these objects are abstracted in module parameters. The script itself is a function parameterized by the reified architecture in Python, obtained as described in 4.1 then mapped to the Coqcots representation. Facets and implementation objects are mapped from Python classes: the class for a facet is a record type; the class for an implementation object is the class of the content object for a component.

The following code written by the reconfiguration developer shows the main tasks in the glue between the Coq-extracted script and Python. The functions

`make_facet`, `make_impl_of_class` and `make_arch` map Python objects to data types extracted from Coq, calling their constructors. First the facets are programmed as classes and mapped to their Coqcots representation (lines 1-9). Then the same is done with implementations (lines 11-16). Then the architecture is reified and mapped to its Coqcots representation (lines 18-21). Last the reconfiguration script is called with all of these objects as parameters (lines 23-25).

```
1  # map facets
2  F = module("F")
3  class dispatcher_provfacet(object):
4      def __init__(self, dispatch):
5          self._dispatch = dispatch
6      def dispatch(self):
7          return self._dispatch
8  F.dispatcher__provfacet = make_facet(dispatcher_provfacet)
9  (F.dispatch,) = namedports_of_facet(F.dispatcher__provfacet)
10
11 # map implementation objects
12 I = module("I")
13 class Dispatcher(object):
14     def dispatch(self, req):
15         # ...
16 I.dispatcher__impl = make_impl_of_class(Dispatcher)
17
18 # map the architecture
19 r = reify(application.receiver)
20 components = dict((d.name,c) for (c,d) in r.iteritems())
21 architecture = make_arch(architecture, components)
22
23 # apply the reconfiguration script
24 R = Reconfiguration.R(RCM, D).R(F).R(I).R(XF).R(XI)
25 R.reconfigure(architecture)
```

The *hotswap* operation is handled specifically. Indeed this operation is expected to dynamically hotswap the implementation of a component, *i.e.*, to apply DSU to the implementation of the content object of the component as depicted in Figure 5. To do so we rely on Pymoult [14], a *dynamic software updating* platform for Python, *i.e.*, reconfiguration at the function-and-object level. Pymoult advocates that, at this level, each reconfiguration may use specific mechanisms in order to accomodate to specific requirements. To follow this recommendation, the *hotswap* operation looks up in a table for a specific hotswapper function provided by the glue code. That hotswapper function can use any of the Pymoult mechanisms to detect / force alterability, update the code, update data, update types and classes, and introspect / reboot / reconstruct thread stacks. The hotswapper function includes all the objects it depends on, including new implementations and code blobs for the component.

In the implementation of the hotswapper, the reconfiguration developer is free to use any update mechanism on a per-operation basis. For instance, if quiescence is required when updating the dispatcher component, the reconfiguration developer can use Pymoult's isClassInAnyStack predicate in order to detect whether its implementation class is active. If the reconfiguration developer prefers the behavior of Java Hotswap (*i.e.*, the reconfiguration occurs immediately; new calls execute the new implementation while ongoing activities complet at the old implementation), she/he has to code the hotswapper accordingly. The following code implements an update function for the latter alternative, the behavior of Java Hotswap. For any component running the `Dispatcher` implementation, and requested to swap to the `Dispatcher_helloSuspended` implementation, it uses Pymoult's class relinking mechanism (hotswap_implementation_class); and it completes when no method of the old `Dispatcher` implementation is on the runtime stacks (isClassInAnyStack).

```
def hotswapper__dispatcher__impl_onlyfile(dispatcher):
    updateDispatcher = BasicUpdate(manager,
          lambda : True,
          lambda : hotswap_implementation_class(dispatcher,
                                    Dispatcher_helloSuspended)
          lambda : not isClassInAnyStack(Dispatcher))
    updateDispatcher.setup()
    updateDispatcher.apply()
    updateDispatcher.wait_update()
register_hotswapper(Dispatcher, Dispatcher,
                    Dispatcher_helloSuspended,
                    hotswapper__dispatcher__impl_onlyfile)
```

4.5. Applying the reconfiguration

The purpose of the step ⑤ of our reconfiguration approach shown on Figure 3 on page 9 is to apply the reconfiguration script obtained during the previous step to the target software system.

As already described, Pycots is depicted by Figure 5. The framework is composed of (1) a `Component` class, which is used to encapsulate components into black boxes, and (2) functions for reconfiguration operations. A component is basically an object that encapsulates its implementation, which is also an object. Each port is a method of this implementation object, which is either injected by the framework (used port) or coded by the developer (provided port). The provided ports are exposed through public *proxy* methods of the component object, which redirect method calls to their destinations.

With this framework, the *link* and *unlink* operations are as simple as field assignment. The two alternatives are: (1) when the used port is bound, the field is assigned to a stub function; (2) otherwise, it refers to the proxy of the provided port it is bound to.

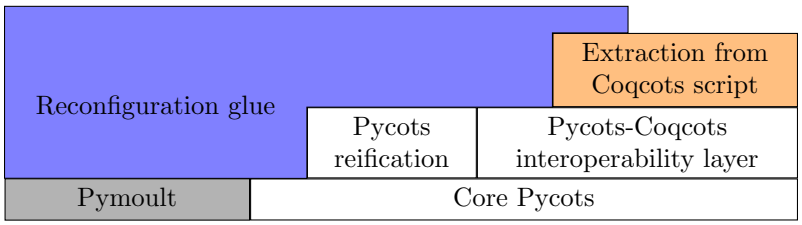

Figure 9: Architecture of Pycots.

The *hotswap* operation is splited in several primitive steps. First, added used ports are injected to the component and removed provided ports are erased. Then, the registered hotswapper function (see Section 4.4) is executed to realize the DSU tasks within the component. Finally, the symmetric of the first step is applied: added provided ports are created and removed used ports are erased from the component. This sequence of operations ensures that at any time, all the provided ports are implemented and all the requirements of the implementation exist as used ports. The primitive port manipulation operations (addition and removal) are implemented using the native ability of Python to dynamically add and remove members to objects.

In summary, the overall structure of the framework is depicted by Figure 9. The core Pycots provides the component concept and primitive operations; the Pycots-Coqcots interoperability layer makes the core functionalities available for code extracted from Coq; Pycots reification introspects a running system to build a representation of its architecture. Here Pymoult is used as a black-box toolset. The script is automatically extracted from the Coqcots reconfiguration script. The reconfiguration glue is written by-hand to gather everything altogether and implement the design choices (especially regarding DSU steps) of the reconfiguration. It is important to notice that the Pycots framework is relieved from runtime verification of constraints, typing and invariants. Indeed, we assume that these issues have been proved with Coq. It results that the core Pycots framework is as tiny as 65 SLOC; the interoperability layer with Coq-extracted code is 205 SLOC; the architecture reification module is 335 SLOC.

5. Summary of the process

In summary, the process depicted on Figure 3 on page 9 is as follow. ① The architecture of the application is first described as a Coq type denoting the set of architectures `a` such that a given proposition `P(a)` holds. This proposition is built as the conjunction of the facts describing all the components and all the bindings in the architecture. It can be automatically generated. ② In Coq, a reconfiguration is a function that applies the reconfiguration operations to an architecture. The Coq proof mode provides an interactive reconfiguration development environment, which generates proof obligations for each precondition

as well as for the preservation of architectural constraints requested by the components. Unprovable subgoals provide hints to the reconfiguration developer to decide what components and implementations could be designed in order to successfully implement the reconfiguration. The return type of the reconfiguration gives a specification of the resulting architecture, describing the components and bindings that are desired as the result of the reconfiguration. Again, Coq's proof mode provides an interactive environment to ensure that the specification holds. In addition to the reconfiguration and its proof of correctness, the Coq code defines the types of the components and of their interfaces. These types can be considered a specification that can be later used when writing the Python code for the component implementations[11]. ③ Once the reconfiguration developer is satisfied, she/he uses Coq's automatic code extraction mechanism to generate the executable reconfiguration script. ④ Following the Coq specification, the reconfiguration developer has to develop Python code for the artifacts that are not programmed in Coq. These artifacts are mainly the implementations of the components as well as the behavior of *hotswap*. ⑤ Last the reconfiguration is executed.

The web page `http://coqcots.gforge.inria.fr` contains the complete source code of Coqcots and Pycots. It also gives instructions to reproduce the web server scenario (reconfiguring from Figure 4 on page 10 to 7 on page 16) used as an example along Section 4.

6. Related work

6.1. Component platforms supporting reconfiguration

OpenCOM [31], Fractal [27] and FraSCAti [32] are well-known component platforms with similar capabilities for managing and reconfiguring component assemblies at runtime. For each component, controller elements are responsible for managing the reconfiguration operations and ensure their safety and consistency. To achieve these guarantees, components can be stopped such that they are led to a *quiescent state*. A typical reconfiguration scenario is (1) stop affected components, (2) change bindings, and (3) (re)start components. Unaffected components are not stopped hence their services remain available during reconfiguration.

When a component A attempts to use a stopped component B, while the behavior is said undefined in the Fractal textual description, most implementations suspend the calling thread until B is restarted. Even if A is not explicitly stopped, its services are unavailable and unavailability propagates back in the architecture. In our case study, at least the dispatcher component must be stopped. But even if the receiver component is not stopped, its thread is suspended as soon as a request is received and until the end of the reconfiguration, thus preventing servicing the requests to serverFile. In practice the whole web

[11] The use of this specification is informal in the current version of Coqcots and Pycots. We consider improving in this regard in future work.

server is therefore disrupted. Alternatively, in the Boyer et al.'s work [6], a consistency invariant requires that mandatory dependencies of started components are bound to started components only: in this case, A must be stopped before B can be. While this approach is better founded, in practice applications are often stopped entirely. In our case study, this invariant forces to stop the receiver component prior to stopping the dispatcher component. With this approach too the whole web server is disrupted. OSGi [33] proposes yet another alternative: the framework-provided `getService` method, used by a bundle to resolve a dependency, informs the bundle when the dependency is missing. To some extent, OSGi supports only optional bindings. While this approach effectively avoids disruption, it is hard to satistify in practice as providing such component implementations that support any unsatisfied dependency is a difficult task. Furthermore, as already stated in this paper, it is not possible to foresee all the ways to avoid disruption when developing the application. Indeed, it highly depends on the currently available resources and the current operating conditions. For example, in our case study, if the current state of the server forbids to use of `serverHello`, it is safe to apply the reconfiguration without modification of the `dispatcher`. Another very different situation would be the case when there are two instances of the dispatcher. In this case, we could elaborate a scheme to update a first instance by modifying the receiver so that it routes all the requests to the second dispatcher and then proceed equally for updating the second dispatcher.

6.2. Verification of reconfiguration

Regarding the verification aspect, none of OpenCOM, Fractal or FraSCAti initially addresses the problem. Léger et al. [34] proposed a model of the semantics of Fractal using Alloy. They focus solely on the global consistency of architectural constraints, which is similar to the verification of the Coqcots invariants as described in Section 4.1. For other aspects, Léger et al. rely on the runtime verification of preconditions during the application of a reconfiguration script, while this verification is achieved statically when developing the reconfiguration with Coqcots.

FracL [35] is another formalization of Fractal for the Focal framework, which in turn relies on Coq. Similarly to Léger et al. they define the invariants of the Fractal component model as well as the semantics of reconfiguration operations as pre-and-postconditions. Then they prove that these reconfiguration operations preserve the invariants. While the invariants from Léger et al. seem more precise (for instance FracL ignores optional ports), the list of invariants is similar. Coqcots is a bit simpler since we do not take into account composite components. In addition to the preservation of invariants, FracL allows specifying reconfiguration scripts by pre-and-postconditions, then develop them using a simple procedural language and prove their conformance. In Coqcots, the language we use is no more no less the Coq language with no restriction.

Still in the context of Fractal, Merle and Stefani [36] formalize the specification of the component model in order to prove that it is consistent. They encode the invariants of the component model as well as the dynamic behavior of the

specified middleware. The Fractal specification is structured in controllers to provide modularity to the model: for instance some components may not have some of the controllers when the corresponding reconfiguration capabilities are not required. In this context, the question thus arises whether the controllers are consistent altogether, *i.e.*, whether their combination does not lead to contradiction. To address this question, Merle and Stefani use the Alloy Analyzer to find at least one instance of the component model. This work is comparable to the proof of consistency of Coqcots described in Section 4.1 on page 13. Merle and Stefani acknowledge that, due to limitations of Alloy and its first-order logic, the behavior of the components cannot be taken into account in the specification of the start/stop operations. While Coq does not suffer the same limitations, we still make this simplification in our specification of the *hotswap* operation. We intend to improve this point in future work.

Mefresa [37] performs a similar task using Coq in the context of the GCM component model, a component model derived from Fractal. While the emphasis is put on the preservation of the invariants, Mefresa formalizes a complete reconfiguration language with an operational semantics, which is rather different from the pre-and-postcondition followed in other works and in Coqcots. While Mefresa uses Coq as a tool to study dynamic reconfigurations, Coqcots uses Coq as the language for reconfiguration hence enabling its extraction mechanism. Following this philosphy leads us leave the specification of Coqcots abstract, such that the extracted scripts are functors parameterized by the concrete Pycots implementation.

Because Boyer et al. [6] aim at providing an automatic reconfiguration protocol, they focus their verification tasks on proving that their protocol conforms to integrity constraints of their component model. Thanks to this, they prove once and for all that any reconfiguration generated by their protocol is correct with respect to this criteria. Our goal is different: we intend to improve service availability as offered by *hotswap* in comparison to start/stop, rather than automate the generation of reconfiguration scripts.

In addition to the formalization of a component model and reconfiguration operations similar to Fractal, Lanoix et al. [38] define a *component substitution* reconfiguration operation. While in appearance this operation is similar to our *hotswap* operation, they differ in the following way: component substitution works at the architectural level, while *hotswap* relies on DSU within a single primitive component. Thus *hotswap* is able to preserve the component state accross the operation, while substitution is not.

The proposal of Bialek and Jul [39] envisions to facilitate the reconfiguration of component-based distributed applications. To take into account the requirement of non-stopping components while reconfiguring, they maintain at the same time the previous and the new versions of a component that needs to be changed. This strategy introduces complexity for managing these elements and may bring up scalability issues, especially when state must be preserved accross component versions. Moreover, the proposal lacks a strong formalism that would ensure important properties throughout the reconfiguration process, such as consistency.

The position paper of La Manna [40] proposes to model the current and new versions of the components using interface automata. These models are then used to automatically generate state transformers, which are functions that map states between the two versions of the interface automata. A state transformer tells when a component can switch from its current version to its new version. This promising approach provides timely, not-disruptive and safe reconfiguration, but the proposal does not consider the implementation aspects.

The work of Andova et al. [41, 42] relies on collaboration models by using the Paradigm coordination language. Each component is described by a state-transition diagram. A collaboration among components follows a set of synchronization constraints that controls the detailed steps of the involved components. As Paradigm is reflective, Andova et al. design a generic component named McPal to migrate the system to new state-transition diagrams (hence changing the behavior of components) and new constraints (hence replacing collaborations) without interrupting any component. Reconfiguration scripts are seen like any other collaboration between the components. Paradigm models and their reconfiguration scripts can be translated to a process algebra where they can be verified. However, the use of state-transition diagrams for modeling the behavior of components is not as expressive as a mainstream language like Python.

In the context of real-time critical embedded systems, Apvrille et al. [43] propose to model a software system and its reconfiguration using a UML profile based on the formal timed process algebra RT-LOTOS. The system is composed with additional components named *observers*. Each observer models a property and can detect erroneous execution traces or missed deadlines because it is synchronized with the system. For reconfiguration, observers can detect service (un)availability and timeliness. Then a reachability analysis checks automatically that no such incorrect situation can occur. However, a reachability analysis expects that the modeled system is a bounded RT-LOTOS model, hence expressivity is restricted.

6.3. Alternatives to proof assistants

Instead of Coq, graph transformations are often used as a formalism for dynamic reconfiguration, like in the work of Heinzemann and Becker [44]. With graph transformation, checking that the resulting architecture conforms to a given graph grammar is as simple as finding a graph morphism. In this paper we address mainly the verification of structural properties, which is left as future work by Heinzemann and Becker [44]. Conversely, Coqcots does not address the verification they perform. The two works are complementary.

The verifications allowed by Coqcots are similar to the ones studied by Cabot et al. [45] in the field of model transformation based on triple graph grammars (TGG). The verification of the TGG rules is translated to the problems of consistency and satisfiability of OCL constraints. They find that the generated constraints are at best EXP-complete, and even undecidable and/or incomplete. Consequently, the verification task is either performed by model-checking when possible (i.e. when the verification problem is bounded) or by a proof assistant.

The direct use of a proof assistant in Coqcots avoids the translation between multiple formalisms.

Automatic generation of the reconfiguration script would make the verification unneeded when the generator is trusted. Arshad [46, 47] and Méhus et al. [48] for instance use PDDL-based planning to generate reconfiguration scripts. The preservation of invariants, to ensure the generation of correct scripts, is encoded in the pre-and-postconditions in the definition of reconfiguration operations. This is similar to Coqcots, and, like we have done in Section 4.1, these pre-and-postconditions must be proved consistent with the desired invariants. Like Boyer et al. [6] did, the generation algorithm must be proved as well. PDDL-based work do not provide these proofs.

While Coqcots has the drawback of needing manual effort, we see a clear advantage advantage. Automatic techniques like PDDL-based planning or model-checking require to anticipate the maximum number of components and all their possible types and implementations. If not, a planner simply answers that no solution exists and gives no hint about the causes of the problem. To illustrate this, consider the reconfiguration of the web server example of Section 4.2 on page 20: automatic techniques have no chance to guess that the developer is able, if needed, to provide an alternative implementation of the dispatcher component that handles some of the requests and enqueues the other requests. If this change of behavior is encoded as a change in the type of the provided services of dispatcher[12], the automatic tool is stuck once again as it cannot guess that the developer could provide a new implementation of receiver. Unless instructed that these implementations exist, an automatic tool is unable to generate the reconfiguration. But to do so, the update developer needs to anticipate all of these implementations. Except for the very general *stopped* implementation like in OpenCOM or Fractal, doing so consists merely in designing the reconfiguration. In contrast with the interactive process of Coqcots, the reconfiguration developer can observe the architecture when stuck. The unprovable proof obligations provide some indications, for instance, about what component implementation might be missing to achieve the desired reconfiguration. The reconfiguration developer can therefore decide to design new implementations or to provide new components in order to complete the reconfiguration. Considering the same example of Section 4.2 on page 20, when the update developer tries to unbind the serveHello port of dispatcher, Coq enforces that she/he proves the architectural constraint assumed by the implementation of dispatcher. Since this constraint requires the serveHello port be bound, the update developer cannot prove it. Observing that the constraint is unprovable, she/he gets a hint that the implementation of dispatcher must be changed in order to weaken the constraint before the port is unbound. Then because this change breaks type consistency of the binding between receiver and dispatcher, which Coq requests

[12] If this is not done, too many details might be abstracted to ensure correctness. In our example, switching dispatcher to an implementation that might suspend the thread must clearly be taken into account by its client components.

to be proved as a precondition of *hotswap*, the update developer gets a hint that a new implementation is needed for receiver in order to accomodate the new behavior of dispatcher.

6.4. Summary

Although the cited approaches support the dynamic reconfiguration of component-based applications, most of them do not address the requirement of service continuity while performing the reconfiguration actions. Unlike the proposals discussed in this section, our approach relies on DSU instead of the conventional start/stop operations. Coqcots focuses on maintaining safety and consistency throughout a reconfiguration process and the Coq proof environment allows to alleviate the complexity of performing DSU operations and enforce properties. Thus, correctness properties and service continuity can be proved by using this approach. Other formalisms could be used to model architectures and reconfigurations, such as graph transformations. Yet it appears that such formalisms still require classical proving techniques to overcome the inherent difficulty of the verification task. While using an interactive approach requires manual effort, it has the advantage to allow dealing with open environments. In such environments the designer is able to imagine new elements to try to keep a high level of quality of service during reconfiguration. When requesting for the proof of architectural invariants, the involved proof assistant gives helpful hints about what new elements might be relevant, as it points out issues. On the other side, automatic tools better suit closed world, where no new component or behavior can be added. In this regard, automatic tools and interactive approaches address two different and complementary situations.

7. Conclusion

Dynamic reconfiguration provides a solution when stopping a component-based software system is not an option. Unlike previous work, our proposal relies on DSU to avoid the conventional start/stop operations over components. Specific component implementations are used during reconfiguration in order to continuously provide the best possible service. These implementations do not need to be anticipated at design time as DSU let us embed them in the reconfiguration. In this paper, we support verification and validation aspects with Coqcots using the Coq proof assistant. By forcing the reconfiguration developer to explicitly reflect any service degradation in the type of the components, Coqcots makes service continuity controllable and provable. We also describe Pycots, an implementation framework developed using the Python language and the Pymoult library. Our case study demonstrates the advantages of the approach.

In the end, our paper contains numerous contributions:

1. A concrete component model *Pycots* for Python. This simple component model offers a complete reflective framework supporting the introspection of a running application to get its current architecture. It also offers a

runtime platform able to execute reconfiguration scripts written in Python. Lastly, Pycots relies on the DSU Pymoult platform to offer hotswap feature of component implementation.

2. An abstract component model *Coqcots* aimed at proving properties on architectures or on manipulations of architectures in the proof assistant Coq. Coqcots supports all the usual operations on architecture: creation or removal of components and bindings, the modification of components including changing their type (the kind of ports they offer) and lastly the hotswapping of the component implementation.

3. A fully bidirectional translation is supported by the reflective feature of Pycots on one side and the extraction facilities of Coq on the other side.

4. A simple tool to automatically manipulate repetitive parts of the Coq proofs. Here we have presented the ability to compress the proof environment by applying a specific coalescing tactic.

5. An extension of the Coq extraction mechanism to enable the generation of Python code. This new backend is intended to be contributed to the Coq community.

6. A method to integrate smoothly the DSU code that needs to be written within the reconfiguration script extracted from Coq. We follow a functor like method to parameterize all the implementation elements. Notice that the DSU function have full access to the Python capabilities but also to the full Pymoult API allowing to write highly tailored and efficient updates.

7. A complete engineering process to help the design of a correct reconfiguration and its application to a running software. The enactment of this process is supported by all the previous contributions.

Our work on Pycots and Coqcots is going on and will continue in mainly four areas:

1. One of the main objectives of the work presented here is to offer a smooth process to be able to validate our approach on more serious applications. The idea here is to try to apply our method to a bigger Python application. Some work has already been done on using Pymoult with Django "a high-level (...) Web framework that encourages rapid development and clean, pragmatic design"[13]. The size of the application would enable to validate the scalability of the proof methods.

2. Since a Coqcots architecture is modeled by a proposition, we envision that we could apply our approach to architectural patterns. As consequence, a reconfiguration developer could develop a reconfiguration script that

[13] https://www.djangoproject.com

would be proved correct for any instance of the architectural pattern. This work will greatly improve the reusability of reconfiguration scripts with high confidence.

3. As we work on DSU mechanism, we intend to explore ways to describe in more details the semantics of the hotswap operation. This objective would enable to prove more fine grain properties on reconfiguration. It would also allow to extract a larger part of the final reconfiguration script. In the end, specifying the DSU part of the glue could be also done within Coq.

4. One feature that Pycots / Coqcots are both lacking is the support for composite component. A component could be an assembly of other components and not only a blackbox element (A group of objects for example in Pycots). This extension requires to revise the proof methodology and to study its impact on the size on the proof environment. Once, the composite feature will be available, we would like to connect the reconfiguration proof engine to another component platform to evaluate its generality. As Fractal is used by a lot of works on reconfiguration of component based applications, it would be a good candidate. Notice that there already exists some DSU functionalities for the execution platform of the main implementation of Fractal: the JVM ([49] or more recently [50]).

[1] J. Kramer, J. Magee, The evolving philosophers problem: Dynamic change management, IEEE Trans. on Software Engineering 16 (11) (1990) 1293–1306. doi:10.1109/32.60317.
URL http://dx.doi.org/10.1109/32.60317

[2] K. Gama, W. Rudametkin, D. Donsez, Resilience in dynamic component-based applications, in: Proc. of the 26th Brazilian Symposium on Software Engineering, SBES 2012, IEEE, Piscataway, NJ, USA, 2012, pp. 191–195. doi:10.1109/SBES.2012.32.

[3] W. Li, QoS assurance for dynamic reconfiguration of component-based software systems, IEEE Trans. on Software Engineering 38 (3) (2012) 658–676. doi:10.1109/TSE.2011.37.

[4] M. Ghafari, P. Jamshidi, S. Shahbazi, H. Haghighi, An architectural approach to ensure globally consistent dynamic reconfiguration of component-based systems, in: Proc. of the 15th ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE'12, ACM, New York, NY, USA, 2012, pp. 177–182. doi:10.1145/2304736.2304765.
URL http://doi.acm.org/10.1145/2304736.2304765

[5] Y. Vandewoude, P. Ebraert, Y. Berbers, T. D'Hondt, Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates, IEEE Trans. on Software Engineering 33 (12) (2007) 856–868. doi:10.1109/TSE.2007.70733.
URL http://dx.doi.org/10.1109/TSE.2007.70733

[6] F. Boyer, O. Gruber, D. Pous, Robust reconfigurations of component assemblies, in: Proc. of the 35th International Conference on Software Engineering, ICSE'13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 13–22.
URL http://dl.acm.org/citation.cfm?id=2486788.2486791

[7] E. Miedes, F. D. Muñoz-Escoí, A survey about dynamic software updating, Tech. Rep. ITI-SIDI-2012/003, Instituto Universitario Mixto Tecnológico de Informática, Universitat Politècnica de València, Valencia, Spain (May 2012).

[8] H. Seifzadeh, H. Abolhassani, M. S. Moshkenani, A survey of dynamic software updating, Journal of Software: Evolution and Process 25 (5) (2012) 535–568. doi:10.1002/smr.1556.
URL http://dx.doi.org/10.1002/smr.1556

[9] J. Purtilo, C. Hofmeister, Dynamic reconfiguration of distributed programs, in: International Conference on Distributed Computing Systems, Arlington, Texas, USA, 1991, pp. 560–571. doi:10.1109/ICDCS.1991.148726.

[10] P. Oreizy, N. Medvidovic, R. N. Taylor, Architecture-based runtime software evolution, in: Proceedings of the 20th International Conference on Software Engineering, ICSE '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 177–186.
URL http://dl.acm.org/citation.cfm?id=302163.302181

[11] J. Kramer, J. Magee, Self-managed systems: An architectural challenge, in: 2007 Future of Software Engineering, FOSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 259–268. doi:10.1109/FOSE.2007.19.
URL http://dx.doi.org/10.1109/FOSE.2007.19

[12] A. Cruz, Official Gmail Blog: Update on today's Gmail outage, http://gmailblog.blogspot.com/2009/02/update-on-todays-gmail-outage.html (Feb. 2009).

[13] C. Giuffrida, A. S. Tanenbaum, Prepare to die: A new paradigm for live update, Technical Report IR-CS-51, Department of Computer Science, Vrije Universiteit, Amsterdam (April 2009).

[14] S. Martinez, F. Dagnat, J. Buisson, Prototyping DSU techniques using Python, in: Proc. of the 5th Workshop on Hot Topics in Software Upgrades, HotSWUp'13, USENIX, Berkeley, CA, USA, 2013.

[15] K.-K. Lau, Z. Wang, Software component models, IEEE Transactions on Software Engineering 33 (10) (2007) 709–724.

[16] OMG, Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 3: CORBA Component Model, http://www.omg.org/spec/CORBA/3.1/Components/PDF (Jan. 2008).

[17] T. Bures, P. Hnetynka, F. Plasil, Sofa 2.0: Balancing advanced features in a hierarchical component model., in: SERA, IEEE Computer Society, 2006, pp. 40–48.

[18] T. Genßler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. Müller, C. Stich, Components for embedded software: the pecos approach, in: CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems, ACM, New York, NY, USA, 2002, pp. 19–26. doi:http://doi.acm.org/10.1145/581630.581634.

[19] OMG, Unified Modeling Language Specification, version 2.0 (July 2003).

[20] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The Fractal component model and its support in Java, Software: Practice and Experience 36 (11-12) (2006) 1257–1284.

[21] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, J. S. Foster, Kitsune: efficient, general-purpose dynamic software updating for C, in: G. T. Leavens, M. B. Dwyer (Eds.), OOPSLA, ACM, 2012, pp. 249–264.
URL http://dblp.uni-trier.de/db/conf/oopsla/oopsla2012.html#HaydenSDHF12

[22] I. Neamtiu, M. Hicks, G. Stoyle, M. Oriol, Practical dynamic software updating for C, in: Proc of the ACM SIGPLAN conference on Programming language design and implementation, PLDI '06, 2006, pp. 72–83. doi:10.1145/1133981.1133991.
URL http://doi.acm.org/10.1145/1133981.1133991

[23] M. Dmitriev, Safe class and data evolution in large and long-lived java[tm] applications, Tech. rep., Sun Microsystems, Inc., Mountain View, CA, USA (2001).

[24] J. Buisson, F. Dagnat, Recaml: execution state as the cornerstone of reconfigurations, in: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming, ICFP '10, 2010, pp. 27–38. doi:10.1145/1863543.1863550.
URL http://doi.acm.org/10.1145/1863543.1863550

[25] J. Arnold, M. F. Kaashoek, Ksplice: automatic rebootless kernel updates, in: European Conference on Computer Systems, 2009, pp. 187–198. doi:10.1145/1519065.1519085.

[26] X. Leroy, Formal verification of a realistic compiler, Commun. ACM 52 (7) (2009) 107–115. doi:10.1145/1538788.1538814.
URL http://doi.acm.org/10.1145/1538788.1538814

[27] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The FRACTAL component model and its support in Java, Software: Practice

and Experience 36 (11-12) (2006) 1257–1284. doi:10.1002/spe.767.
URL http://dx.doi.org/10.1002/spe.767

[28] P. J. Hayes, The frame problem and related problems in Artificial Intelligence, Tech. rep., Stanford University, Stanford, CA, USA (1971).

[29] P. Brada, Enhanced type-based component compatibility using deployment context information, Electronic Notes in Theoretical Computer Science 279 (2) (2011) 17–31. doi:10.1016/j.entcs.2011.11.009.
URL http://dx.doi.org/10.1016/j.entcs.2011.11.009

[30] D. Scott, A system of functional abstraction, lectures delivered at University of California, Berkeley, Cal., 1962/63. Photocopy of a preliminary version, issued by Stanford University (sep 1963).

[31] P. Pissias, G. Coulson, Framework for quiescence management in support of reconfigurable multi-threaded component-based systems, IET Software 2 (4) (2008) 348–361. doi:10.1049/iet-sen:20070046.

[32] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, J.-B. Stefani, A component-based middleware platform for reconfigurable service-oriented architectures, Software: Practice & Experience 42 (5) (2012) 559–583. doi:10.1002/spe.1077.
URL http://dx.doi.org/10.1002/spe.1077

[33] A. L. Tavares, M. T. Valente, A gentle introduction to osgi, SIGSOFT Softw. Eng. Notes 33 (5) (2008) 8:1–8:5. doi:10.1145/1402521.1402526.
URL http://doi.acm.org/10.1145/1402521.1402526

[34] M. Léger, T. Ledoux, T. Coupaye, Reliable dynamic reconfigurations in a reflective component model, in: Proceedings of the 13th International Conference on Component-Based Software Engineering, CBSE'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 74–92. doi:10.1007/978-3-642-13238-4_5.
URL http://dx.doi.org/10.1007/978-3-642-13238-4_5

[35] M. Simonot, V. Aponte, A declarative formal approach to dynamic reconfiguration, in: Proceedings of the 1st International Workshop on Open Component Ecosystems, IWOCE '09, ACM, New York, NY, USA, 2009, pp. 1–10. doi:10.1145/1595800.1595802.
URL http://doi.acm.org/10.1145/1595800.1595802

[36] P. Merle, J.-B. Stefani, A formal specification of the Fractal component model in Alloy, Research Report RR-6721 (2008).
URL https://hal.inria.fr/inria-00338987

[37] N. Gaspar, L. Henrio, E. Madelaine, Bringing coq into the world of gcm distributed applications, International Journal of Parallel Programming 42 (4) (2014) 643–662. doi:10.1007/s10766-013-0264-7.
URL http://dx.doi.org/10.1007/s10766-013-0264-7

[38] A. Lanoix, O. Kouchnarenko, Component substitution through dynamic reconfigurations, in: B. Buhnova, L. Happe, J. Kofron (Eds.), Proceedings 11th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2014, Grenoble, France, 12th April 2014, Vol. 147 of EPTCS, 2014, pp. 32–46. doi:10.4204/EPTCS.147.3.
URL http://dx.doi.org/10.4204/EPTCS.147.3

[39] R. Bialek, E. Jul, A framework for evolutionary, dynamically updatable, component-based systems, in: Proc. of the Workshops at the 24th International Conference on Distributed Computing Systems, ICDCS 2004, IEEE, USA, 2004, pp. 326–331. doi:10.1109/ICDCSW.2004.1284050.

[40] V. P. L. Manna, Local dynamic update for component-based distributed systems, in: Proc. of the 15th ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE'12, ACM, New York, NY, USA, 2012, pp. 167–176. doi:10.1145/2304736.2304764.
URL http://doi.acm.org/10.1145/2304736.2304764

[41] S. Andova, L. P. J. Groenewegen, J. Stafleu, E. P. de Vink, Formalizing adaptation on-the-fly, Electronic Notes in Theoretical Computer Science 255 (2009) 23–44. doi:10.1016/j.entcs.2009.10.023.
URL http://dx.doi.org/10.1016/j.entcs.2009.10.023

[42] S. Andova, L. P. J. Groenewegen, E. P. de Vink, Distributed adaption of dining philosophers, in: Proc. of the 7th International Conference on Formal Aspects of Component Software, FACS'10, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 125–144. doi:10.1007/978-3-642-27269-1_8.
URL http://dx.doi.org/10.1007/978-3-642-27269-1_8

[43] L. Apvrille, P. De Saqui-Sannes, P. Sénac, C. Lohr, Verifying service continuity in a dynamic reconfiguration procedure: Application to a satellite system, Automated Software Engineering 11 (2) (2004) 167–191. doi:10.1023/B:AUSE.0000017742.47984.6c.
URL http://dx.doi.org/10.1023/B:AUSE.0000017742.47984.6c

[44] C. Heinzemann, S. Becker, Executing reconfigurations in hierarchical component architectures, in: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering, CBSE '13, ACM, New York, NY, USA, 2013, pp. 3–12. doi:10.1145/2465449.2465452.
URL http://doi.acm.org/10.1145/2465449.2465452

[45] J. Cabot, R. Clarisó, E. Guerra, J. de Lara, Verification and validation of declarative model-to-model transformations through invariants, J. Syst. Softw. 83 (2) (2010) 283–302. doi:10.1016/j.jss.2009.08.012.
URL http://dx.doi.org/10.1016/j.jss.2009.08.012

[46] N. Arshad, Automated dynamic reconfiguration using ai planning, in: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, ASE '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 402–405. doi:10.1109/ASE.2004.16.
URL http://dx.doi.org/10.1109/ASE.2004.16

[47] N. Arshad, D. Heimbigner, A. L. Wolf, Deployment and dynamic reconfiguration planning for distributed software systems, in: Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence, ICTAI '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 39–.
URL http://dl.acm.org/citation.cfm?id=951951.952269

[48] J.-E. Méhus, T. Batista, J. Buisson, ACME vs PDDL: support for dynamic reconfiguration of software architectures, in: 6ème édition de la Conférence Francophone sur les Architectures Logicielles (CAL 2012), Montpellier, France, 2012, pp. 48–57.
URL https://hal.archives-ouvertes.fr/hal-00703176

[49] S. Subramanian, M. Hicks, K. S. McKinley, Dynamic software updates: A vm-centric approach, in: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, ACM, New York, NY, USA, 2009, pp. 1–12. doi:10.1145/1542476.1542478.
URL http://doi.acm.org/10.1145/1542476.1542478

[50] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, J. L, Low-disruptive dynamic updating of java applications, Information and Software Technology 56 (9) (2014) 1086 – 1098, special Sections from Asia-Pacific Software Engineering Conference (APSEC), 2012 and Software Product Line conference (SPLC), 2012. doi:http://dx.doi.org/10.1016/j.infsof.2014.04.003.
URL http://www.sciencedirect.com/science/article/pii/S0950584914000846